Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Research Report
RR-97-05

Finding Regions for Local Repair in Hierarchical Constraint Satisfaction

Harald Meyer auf'm Hofe

December 1997

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
67608 Kaiserslautern, FRG
Tel.: + 49 (631) 205-3211
Fax: + 49 (631) 205-3210

Stuhlsatzenhausweg 3
66123 Saarbrücken, FRG
Tel.: + 49 (681) 302-5252
Fax: + 49 (681) 302-5341

Finding Regions for Local Repair in Hierarchical Constraint Satisfaction

Harald Meyer auf'm Hofe

DFKI-RR-97-05

This report summarizes research work that has been presented at PACT-97 and the CP-97 workshop on theory and practice of dynamic constraint satisfaction.

This work has been supported by a grant from The Federal Ministry of Education, Science, Research, and Technology (FKZ ITWM-9702) and by a grant of the foundation “Rheinland-Pfalz für Innovation” (grant-no. 836-38 62 61 /98).

ISSN 0946-008X

Finding Regions for Local Repair in Hierarchical Constraint Satisfaction

Harald Meyer auf'm Hofe

Abstract

Algorithms for solving constraint satisfaction problems (CSP) *have been successfully applied to several fields including scheduling, design, and planning. Latest extensions of the standard CSP to constraint optimization problems (COP) additionally provided new opportunities for solving several problems of combinatorial optimization more efficiently. Basically, two classes of algorithms have been used for searching* constraint satisfaction problems (CSP): *local search methods and systematic tree search extended by the classical constraint-processing techniques like e.g.* forward checking *and* backmarking. *Both classes exhibit characteristic advantages and drawbacks. This report presents a novel approach for solving constraint optimization problems that combines the advantages of local search and tree search algorithms which have been extended by constraint-processing techniques. This method proved applicability in a commercial nurse scheduling system as well as on randomly generated problems.*

1 Introduction

Algorithms for solving *constraint satisfaction problems (CSP)* have been successfully applied to several fields including scheduling, design, and planning. A CSP implies the task of labeling each variable of a given variable set with a value of a certain domain. Constraints state restrictions on combinations of some variables' labels. The solution of a CSP is a labeling complying with all constraints.

Latest extensions of the standard CSP to constraint optimization problems (COP) [3] additionally provided new opportunities for solving several combinatorial optimization problems more efficiently. Yet, several notions of soft constraints have been investigated more detailed. On the one hand, these notions provide distinguished facilities for representing preference, assumption, and cost measures by so called *soft constraints.* On the other hand, certain techniques of constraint processing are directly related to special formalisms of *soft* constraints. This report puts a focus on *soft* and *crisp* constraints. A *soft* constraint is preferred but not required to be satisfied by a solution. Hence, a problem comprising soft constraints is an optimization problem, where a solution satisfies all *hard* or compulsory constraints and complies additionally with the soft constraints in an optimal way. A *crisp* constraint is either satisfied or violated. *Crisp* constraints cannot be satisfied partially by a solution like so called *fuzzy* constraints [6].

Typically, soft and crisp constraints have special properties characterizing their relative importance. The solution is required to satisfy *as important soft constraints* as possible. A constraint's priority [7] is a formalism for representing importance which is appropriate to state a categorical measure of preference respectively believe. A constraint of larger priority is defined to be more important than all constraints of smaller

priority together. In contrast, constraint weights [8] state a gradual measure of importance. A labeling of all variables is told to be a solution iff the weight sum of its constraint violations is as small as possible. Hence, violating many less important constraints can be worse than violating a few very important constraint.

Hierarchical constraint satisfaction (HCSP) [15, 16] is a formalism for integrating both aspects of a constraint's importance and is inspired by *hierarchical constraint logic programming (HCLP)* [5]. Soft constraints are grouped into hierarchy levels which are closely related to constraint priorities. Arbitrary formalisms can be used to define the relative importance of a constraint within one hierarchy level.

Basically two classes of algorithms have been applied to solve problems comprising soft constraints: local search methods [18, 22, 25] and systematic *branch&bound* search extended by several constraint-processing techniques [8]. Both paradigms exhibit characteristic advantages and drawbacks. In theory, the *branch& bound* algorithm is guaranteed to terminate with an optimal solution. However, tree search algorithms retract early decisions only after searching large portions of the search space exhaustively. As a consequence, minor differences in CSPs can result into completely different run time behavior — especially if the number of variables is larger. In contrast, local search procedures can return a result at any time. However, this result is of a questionable quality. Especially, proving the optimality of a result is not possible by use of these algorithms.

Contribution: This report presents a new approach for solving constraint optimization problems that combines the advantages of local search and tree search algorithms which have been extended by constraint processing techniques. The first contribution is the introduction of a scheme for repair-based search comprising three steps. After computation of a (possibly randomly chosen) initial labeling of the variables according to prospective processing results of hard and soft constraints conduct the following steps iteratively:

1. Choose a region (a set of variables) in the problem and reset the variables in this region. At this point, the method typically needs to be tailored to the current application.
2. Propagate all constraints between reset variables and the persistent labels. Optimize the labels in the reset region using an extended *branch&bound*.
3. Lock the exhaustively searched region by temporary constraints.

While this contribution may be considered as a specific instance of folklore in the field of constraint processing, the second contribution certainly is a novel extension of this scheme turning it into a *generic and exhaustive* search algorithm. In this generic family of algorithms, step 1 enumerates all regions in the given problem whose labels may cause a constraint violation. This enumeration is filtered with respect to a generic control strategy without any reference to application specific search control knowledge.

Several experiments have been conducted in order to prove the relevance of the contributions. Both, various extensions of the *branch&bound* algorithm and several local search strategies, have been implemented efficiently in the ConPlan C++ program library. This library has been designed for solving combinatorial problems in applications and supporting research on search algorithms. By use of this implementation, an instance of the proposed repair-based search proved its applicability in the commercial nurse scheduling system SIEDAplan [15]. The proposed complete enumeration of regions for local repair [16] has been assessed by empirical studies on randomly generated problems.

Structure: This report is structured as follows: Section 2 describes methods for representing and solving constraint optimization problems including the contributed scheme for repair-based search. Section 3 exemplifies contribution one presenting the SIEDAplan system which treats nurse scheduling as a constraint optimization problem. Section 4 explains the contributed complete method for enumerating regions of local repair. Section 5 presents an empirical evaluation of this method. In conclusion, the presented results are summarized .

2 Constraint Optimization

Before explaining formalisms for specifying constraint optimization problems and discussing previously proposed algorithms for solving these problems, some basic definitions are given here in order to clarify notation.

Definition 1 *A CSP is a tuple* (V, D, C) *where*

- V *is a set of variables,*
- D *is the domain of the variables, a set of values which can be assigned to the variables, and*
- C *is a set of constraints, where each* $c \in C$ *is defined by:*
  - $V(c) \in V$ *is a set of variables which are* directly affected *by* c*.*
  - *The extension of* c*,* $\mathrm{ext}(c)$*, is a set of labelings of all variables in* $V(c)$ *with values of* D *which comply with* c*.*

Let $l = \{v_1 \leftarrow d_1, \ldots, v_n \leftarrow d_n\}$ *be a labeling of all variables in* $V' = \{v_1, \ldots, v_n\}$ *and* $V'' \subseteq V' \subseteq V$*. Then,* $l \downarrow_{V''} = \{v_i \leftarrow d_i \mid v_i \in V''\}$ *denotes the selection of labels which concern the variables in* V''*. A labeling* l *complies with constraint* c *iff* $l \downarrow_{V(c)} \in \mathrm{ext}(c)$*.* $\bar{C}(l) = \{c \in C \mid l \downarrow_{V(c)} \notin \mathrm{ext}(c)\}$ *denotes the set of constraints being violated by* l*. A labeling* l *of all variables in* V *is a solution of* $CSP = \{V, C, D\}$ *iff*

$$\begin{aligned} l \in \mathrm{ext}(C) &\iff \forall c \in C : l \downarrow_{V(c)} \in \mathrm{ext}(c) \\ &\iff \bar{C}(l) = \{\} \\ &\iff l \in \bowtie_{c \in C} \mathrm{ext}(c). \end{aligned}$$

Hence, a solution of a CSP consists of labelings of all variables which comply with all constraints.

2.1 Formalizations of Constraint Optimization Problems

In contrast to CSPs, solutions of *constraint optimizations problems (COP)* are only required to comply with as important constraints as possible. Presupposing crisp constraints, the most general representation of importance is a partial ordering $\succ$ on constraint sets where $C' \succ C''$ means intuitively: *The constraints in* C' *are more important than the constraints in* C'' [14]. The following definition of COPs reflects this idea.

Definition 2 *A* $COP = (V,\ D,\ C,\ \succ)$ *extends a* $CSP = (V,\ D,\ C)$ *by a preference ordering* $\succ$ *which is a partial ordering among subsets of* C*.*

A labeling l *of all variables in* V *is a solution of a COP iff there is no labeling* $l' \neq l$ *with* $\bar{C}(l) \succ \bar{C}(l')$*.*

Several integrating views on soft constraints have been proposed [21, 4, 3] in order to achieve a better understanding of constraint processing techniques. Contrarily, this view aims at making the formalization of real world problems as COPs easier. The meaning of preference orderings $\succ$ can be explained relatively easily even to non-experts in constraint processing. Additionally, this framework enables soft constraints to avoid global scales of merit like weights or priorities. This ability is especially important to represent optional requirements in scheduling, configuration, and design. In these fields, preference among requirements on a configuration or a design is typically given only relative to the other requirements. Each requirement on a configuration or a design is typically represented by a constraint. Soft constraints are used to represent optional requirements. Thus, for each pair of constraints c' and c'' it is only given, whether one is more important than another ($c' >_c c''$ or $c'' >_c c'$), or they are not comparable at all. A preference ordering $\succ_c$ of the following kind reflects this situation because it is defined due to the semantics of the partial ordering $>_c$.

Definition 3 *A problem of partially ordered constraints* $(V,\ D,\ C,\ >_c)$ *is equivalent to a* $COP_c = (V,\ D,\ C,\ \succ_c)$ *with*

$$C' \succ_c C'' \quad \textit{iff} \quad \exists c' \in C' \setminus C'' : \forall c'' \in C'' \setminus C' : c' >_c c'' \textit{ and } \forall c'' \in C'' \setminus C' : \exists c' \in C' \setminus C'' : c' >_c c''.$$

Roughly spoken, C' is more important than C'' if for each constraint in C'' there is a more important one in C'.

In contrast, the common formalizations of COPs use numbers to define the importance of a constraint. For instance, the formalism of weighted constraints [8] introduces weights $\omega(c)$ for each constraint $c \in C$.

Definition 4 *A problem of weighted constraints* $(V,\ D,\ C,\ \omega)$*, where* ω *maps a real number to each constraint, is equivalent to a* $COP_\omega = (V,\ D,\ C,\ \succ_\omega)$ *with*

$$C' \succ_\omega C'' \Longleftrightarrow \sum_{c' \in C'} \omega(c') > \sum_{c'' \in C''} \omega(c'').$$

The sum of the weights determines which constraint set is preferred to be satisfied.

In problems of prioritized constraints [7], the most important violated constraint determines the merit of a labeling.

Definition 5 *In problems of prioritized constraints* $(V,\ D,\ C,\ p)$*, each constraint* c *has a priority* $p(c) \in\ ]0:1]$*.* $(V,\ D,\ C,\ p)$ *is equivalent to a* $COP_p = (V,\ D,\ C,\ \succ_p)$ *with*

$$C' \succ_p C'' \textit{ iff } max\{p(c') \mid c' \in C'\} > max\{p(c'') \mid c'' \in C''\}.$$

The HCLP scheme [5] introduced lexical combinations of preference criteria. Several preference criteria are treated one by one in a certain order due to their importance. The most important criterion is considered first. If one of the two sets is preferred according to this criterion, this set will also be preferred due to the whole list. Otherwise, the next criterion is considered. This procedure is repeated until either the importance of both sets can be distinguished or all criteria have been treated but failed. The idea of constraint hierarchies is to associate each of these criteria with hierarchy levels. Lower hierarchy levels will only be considered, if two constraints cannot be distinguished according to more important criteria.

Definition 6 *A* hierarchical constraint satisfaction problem $HCSP = (V,\ D,\ H)$ *is declared dividing the set of constraints* C *into a finite number of hierarchy levels* C_i *such that each constraint is in exactly one hierarchy level.* C_0 *is thought to comprise compulsory constraints. Each hierarchy level* C_i *except* C_0 *is associated with a preference ordering* $\succ_i$ *on subsets of* C_i*. If* $C' \cap C_i \succ_i C'' \cap C_i$ *holds true then* C' *is said to be preferred to* C'' *with respect to hierarchy level* i*. A* $HCSP = (V,\ D,\ H)$ *is equivalent to a* $COP_H = (V,\ D,\ C,\ \succ^1)$ *with*

$$C' \succ^i C'' \quad \textit{iff} \quad C' \cap C_i \succ_i C'' \cap C_i \\ \textit{or } (\neg(C'' \cap C_i \succ_i C' \cap C_i) \textit{ and } C' \succ^{i+1} C'').$$

The semantics of hierarchy levels is closely related to constraint priorities.

Property 1 *Presupposing* $p(c) = \frac{1}{i+1}$ *for each constraint* $c \in C_i$ *of level* i *in an arbitrary hierarchy* H *then, obviously, the following holds true:*

$$C' \succ_p C'' \Longrightarrow C' \succ_H C''.$$

Hence, a constraint hierarchy can be considered as a system of prioritized constraints with some additional preference relations.

A link between hierarchical constraints and weighted constraints can be established using constraint weights as a preference ordering within each hierarchy level. This is the kind of preference orderings that is currently supported by the introductory mentioned ConPlan system. The following property shows a way to derive *global* constraint weights from the weights which have been defined within a hierarchy.

Property 2 *A* $HCSP = (V,\ D,\ H)$ *with* $H = (\{\},\ (C_1, \succ_\omega), \ldots,\ (C_h, \succ_\omega))$ *is equivalent to a* $COP = (V,\ D,\ \bigcup_{i=0}^{h} C_i,\ \succ_{\omega'})$ *with*

$$\omega'(c) = \omega(c) \textit{ iff } c \in C_h,$$
$$\omega'(c) = (\omega(c)+1) \sum_{j=i+1}^{h} \sum_{c' \in C_j} \omega'(c') \textit{ iff } c \in C_i \textit{ with } i < h.$$

Hence, in this kind of constraint hierarchies the importance of constraint violations can be represented by a scalar weight which is important for the diagrams given in section 5.

2.2 Solving Constraint Optimization Problems

This section gives a short introduction into the search algorithms which have been implemented in the C++ program library ConPlan at DFKI. The following three sections briefly describe possible extensions of the *branch&bound* algorithm for solving COPs, well known algorithms of local search, and a new scheme for repair-based search.

2.2.1 Extending *Branch&Bound* Search

The ConPlan library provides certain constraint processing extensions to the common *branch&bound* algorithm that can be explained according to the structure of *branch&bound* algorithms as given in figure 1. The task is to find a labeling

$$s = (v_1 \leftarrow d_1, \ldots, v_n \leftarrow d_n)$$

branch&bound$(s, l, b, \delta, COP = (V, D, C, \succ))$

1. *if* $|l| = |s|$ *and* $b \succ \delta$ *then*
   begin $b := \delta$; $s := l$; *return* s; *end;*
2. *if* $|l| = |s|$ *then*
   return s;
3. *choose a variable* $v \in V$ *that is not labeled by* l;
4. *forall values* $d \in D$ *in the domain of variable* v *do begin*
   (a) $\delta_{local} = \{c \in C | c$ *is inconsistent with* $v \leftarrow d\}$
   (b) *if* $\delta_{local} \cap C_0 = \emptyset$ *and* $\delta \cup \delta_{local} \not\succ b$ *then*
   $$s \leftarrow branch\&bound(s, l \cup \{v \leftarrow d\}, b, \delta \cup \delta_{local}, COP);$$
5. *return* s;

initial call:
branch&bound$(\emptyset, \emptyset, C, \emptyset, COP)$

Figure 1: Branch&bound *algorithms.*

for the variables $v_1, \ldots, v_n \in V$ with values $d_1, \ldots, d_n \in D$, such that s violates a set of constraints b that is minimal according to $\succ$. The algorithm completes a partial labeling l step by step. For each variable all available values are considered (row 4) as labels. The algorithm checks for new constraint violations (row 4a) and maintains the set δ that holds the constraints violated by labeling l. Then, the new branch in the search tree is expanded (row 4b) considering all complete labelings comprising l and $v \leftarrow d$. If all variables have been labeled by l and l violates less important constraints than s, then l will be taken as new assumption for the solution (row 1). Thus, s always holds the best labeling found so far. Hence, if all labelings have been searched s will hold an optimal solution. To be more efficient, the algorithm expands only a new branch of the search tree if l extended by $v \leftarrow d$ satisfies all hard constraints and violates less important constraints than stored in the bound b (row 4b).

Figure 1 differs from common representations of the *branch&bound* [20, 8] only in one point: A labeling's merit is not stored as a real number but as a set of violated constraints (distance δ and bound b). Two labelings are compared by the preference ordering $\succ$ instead of using the natural ordering of reals.

Certain extensions of the *branch&bound* have been proposed employing adoptions of constraint processing techniques in order to increase efficiency.

Pruning: In row 4a the term "inconsistent with" can have several concrete meanings. Naive implementations only check constraints between labeled variables. In contrast, *forward checking* with domains of yet unlabeled variables is possible either with hard and soft constraints [8]. On the one hand, these procedures prune domains of unlabeled variables in advance. On the other hand, an optimistic estimate $\delta \cup \delta_{local}$ of the best labelings merit in branch $v \leftarrow d$ is computed to reduce branching in row 4b of the algorithm. Additionally, arc-consistency with hard constraints [12] can be maintained after each assignment for the same purpose.

For prioritized constraints, an arc-consistency algorithm has been proposed which labels each value in a domain with a compatibility index [23]. The index i of value v represents an optimistic estimate of the priority level that can be completely satisfied assigning v, i.e. the algorithm proves that not all constraints of priority $p < i$ can be satisfied in the current branch. Such compatibility values can be used as an optimistic

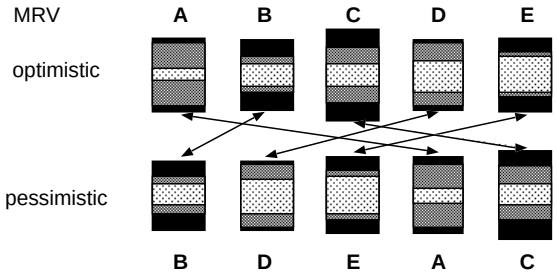


Figure 2: *Two extensions of* minimal remaining values *for HCSPs.*

estimate of the best labeling's merit in the current branch to reduce branching in row 4b. Referring to property 1, the algorithm for arc-consistency with prioritized constraints is appropriate to improve searching constraint hierarchies, as well [17].

Value ordering: As presented in the previous paragraph, prospective constraint processing of soft constraints results in an optimistic estimate of the solution quality that can be achieved in branch $v \leftarrow d$. This estimate can be exploited to consider the values first in row 4 that probably are part of a sufficient solution. The values which are known to cause more important constraint violations are tried later on.

Variable ordering: Several heuristics have been proposed to inform the choice of variables in row 3. *Maximal width* (static) [24] and *minimum remaining values* (dynamic) are the mostly used strategies for variable reordering. The *minimum remaining values (MRV)* heuristic is especially useful if prospective constraint processing is done [2]. When the tree-search algorithm determines which variable to label next, it chooses the one with the minimal number of values compatible with the previous assignments. The goal of the MRV heuristic is to label strongly interfering variables consecutively. Searching for optimal solutions of constraint optimization problems, the MRV heuristic is even more important because variables with smaller domains are labeled first. This strategy decreases the number of nodes that have to be visited, because the larger domains are explored deeper in the search-tree.

The strong relation between prospective constraint processing and the *minimum remaining values* heuristic suggests to exploit *forward checking* of soft constraints and compatibility values, as well. When searching constraint hierarchies, not only the number of values consistent with C_0 is considered but also the number of values consistent with $C_0 \cup C_1$ (compatibility level at least 2), $C_0 \cup C_1 \cup C_2$ (compatibility level at least 3) and so on. These domain sizes can of course be considered in different orders:

MRV pessimistic: At first select all variables with the smallest number of values consistent with compulsory constraints in C_0. Break ties by the number of values consistent with $C_0 \cup C_1$ and so on.

MRV optimistic: This heuristic considers the domain sizes in reverse order: From the number of values consistent with the whole hierarchy to the number of values consistent with C_0.

As the example in Figure 2 shows, the optimistic and the pessimistic way of variable reordering can behave differently due to the satisfiability of each hierarchy level. Each box represents a variable's domain. The shadings indicate the different compatibility levels in the domains — the brighter the region the more hierarchy levels are consistent with the values represented by this region.

Part	*mincon*$(COP = (V, D, C, \succ))$	*minconwalk*$(p, COP = (V, D, C, \succ))$
1.	*compute an initial labeling* l *of all variables in* V*;* *set* $V_{visited} := \{\}$*;*	*compute an initial labeling* l *of all variables in* V*;*
2.	*choose an arbitrary* $(v \leftarrow d) \in l$ *with* $v \notin V_{visited}$*;* *set* $l' := l \setminus \{(v \leftarrow d)\}$ $V_{visited} := V_{visited} \cup \{v\}$*;*	*choose an arbitrary* $(v \leftarrow d) \in l$*;* *set* $l' := l \setminus \{(v \leftarrow d)\}$*;*
3.	*find* $(v \leftarrow d')$ *with minimal* $\bar{C}(l' \cup \{(v \leftarrow d')\})$*;* *if* $\bar{C}(l) \succ \bar{C}(l' \cup \{(v \leftarrow d')\})$ *then begin* $l := l' \cup \{(v \leftarrow d')\}$*;* $V_{visited} := \{\}$*; end;*	*choose an arbitrary* $(v \leftarrow d')$*;* *if* $\bar{C}(l) \succ \bar{C}(l' \cup \{(v \leftarrow d')\})$ *or with probability* p *do* $l := l' \cup \{(v \leftarrow d')\}$*;*
4.	*if time exceeded or* $\bar{C}(l)$ *is small enough* *then return* l*;*	*if time exceeded or* $\bar{C}(l)$ *is small enough* *then return* l*;*
5.	*if* $V_{visited} = V$ *then goto 1 else goto 2;*	*goto 2;*

Figure 3: *The algorithms* mincon *and* minconwalk

The optimistic procedure is equivalent to the MRV procedure in an under-constrained system, where each constraint is hard. However, in over-constrained problems this heuristic often misleads, because consistency with a possibly unsatisfiable constraint set is considered as a main criterion for variable ordering. Hence, optimistic MRV is used only in conjunction with arc-consistent compatibility levels, i.e. the search procedure looks deeply forward into the search space.

2.3 Local Search: mincon and minconwalk

Yet, two closely related approaches of local search have been applied to COPs: *minimizing conflicts* (mincon) [18] and minconwalk [25]. The latter is inspired by the GSAT algorithm [22] which is used for solving SAT problems. Figure 3 shows both algorithms which rely on the same idea of successive improvement of a single labeling — in contrast to tree search algorithms which conduct reasoning on branches of a search tree. Consequently, local search algorithms exhibit the same structure which consists of five parts.

1. Initialization: First, an initial labeling is computed.
2. Possible local modifications of the current labeling are considered.
3. Acceptance: This local modification is either accepted or rejected depending on certain conditions.
4. Termination: The algorithms cannot determine whether the current labeling is really optimal or not. Hence, criteria for terminating search have to be given externally e.g. by a time bound or a bound on constraint checks etc.
5. The algorithms return to step 2 in order to conduct the next improvement step.

Local search always has the problem of avoiding local minima, where none of the possible local improvement steps in part 2 leads to an immediate improvement of the current labeling but the current labeling is not optimal. mincon and minconwalk employ different strategies to cope with this problem:

mincon: If none of the local modification steps leads to an improvement then a new initial labeling is generated by (more or less) randomly chosen assignments of labels to variables. Therefore, in Figure 3 a set of variables $V_{visited}$ is maintained

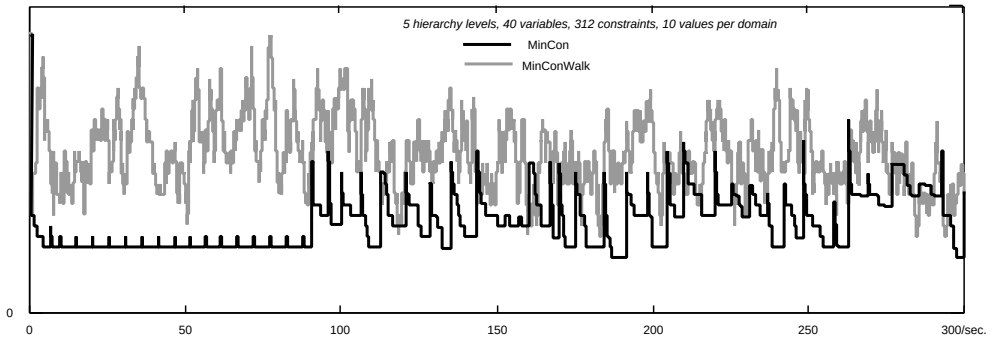


Figure 4: *Improvement of current labeling:* mincon *and* minconwalk.

which includes all the variables that have been visited after the last successful improvement. If all variables have been visited in row 5 than the algorithm looks for a new initial labeling.

minconwalk: This algorithm accepts with a probability of p that the current labeling becomes worse than the best yet found labeling in order to escape from local minima.

Figure 4 shows the different behavior of both algorithms. A randomly generated HCSP is searched by both algorithms. The curves report the weight of constraints which are violated by the current labeling according to property 2. The quality of the current labeling maintained by the minconwalk algorithm goes constantly up and down because worse labelings are accepted with a probability of 0.4. The curve of the mincon algorithm shows that the current labeling is improved continuously until a local minimum is reached. Peaks in this performance curve indicate situations where a labeling is generated from scratch because a local minimum has been detected. This new labeling is of course mostly worse than the local minimum the algorithm tries to escape from.

In the ConPlan library, methods of constraint propagation and variable reordering are used to improve the quality of the initial labeling. Violations of soft constraints, which have been detected in advance, are used to label the variables with values which are likely to be parts of good solutions. Labels implying the same degree of constraint violation are distinguished randomly.

2.4 Iterative Improvement

Figure 5 presents a scheme for local search which may be considered as a generalization of mincon. However, in contrast to the simple mincon method, this more general scheme of iterative improvement steps over local minima by conducting more than one change of the current labeling l within a single step of repair. Therefore, a procedure *choose-bad-region* is used to determine a set of variables whose labels will be changed in the following improvement step (row 2). The constraint graph of the problem, the domains of the variables[1], the current labeling, and the set of violated constraints are useful parameters of this procedure. This improvement step is conducted by a variant of the *branch&bound* in row 4. Only improvements of l are acceptable (row 5). Hence, it is possible to use the current set of constraints violated by l as initial bound and employ several kinds of constraint propagation in order to improve performance of each

[1]Maybe as obtained from arc-consistency preprocessing that has been done before starting the search.

iterative-improvement$(V, D, C, \succ)$

1. *Compute an initial labeling* l *of all variables in* V*; store the violated constraints in* δ*;*
2. $V' =$ *choose-bad-region*$(V, C, \succ, l, \delta)$*;*
3. *if* $V' = \emptyset$ *then go to 8;*
4. *unassign the variables in* V'*, propagate all constraints* c *with* $V' \cap V(c) \neq \{\} \wedge V' \cap V(c) \neq V(c)$ *in order to detect incompatibilities with persistent labelings, and run* branch&bound *on the variables in* V' *with* δ *as initial bound* b*;*
5. *if* $\delta \succ b$ *(the new bound* b *comprises less important constraints than* δ*), then* $\delta := b$ *and assign the new labels to* l*;*
6. *add temporary hard constraint*
$$\bigvee_{v \in V \setminus V'} v \neq l\downarrow_v$$
requiring all labelings visited from now on to differ in at least one assignment from the labelings visited by the last run of the branch&bound*;*
7. *go to 2;*
8. *remove temporary constraints. return* l *as result.*

Figure 5: *Searching by iterative improvement.*

improvement step. *Branch&bound* searches all possible labelings of V' exhaustively. Consequently, further improvement steps have to consider a change in $V \setminus V'$ in order to prevent the search algorithm from visiting labelings more than once. This condition is enforced in row 6 by an additional temporary constraint.

This scheme for decomposing constraint problems demands constraint models where usually a relatively small number of changes enables a global improvement of local minima. Hence, local improvement is a *structural method* whose applicability depends strongly on the structure of the problem's constraint model. Moreover, the problem of finding out where to repair the current labeling remained untouched, yet. This report introduces two solutions of this question:

1. A heuristic version of *choose-bad-region* is given in section 3 that controls search in a commercial nurse scheduling system and, therefore, proves applicability of this search control scheme.
2. An exhaustive algorithm for *choose-bad-region* is presented in section 4 in order to turn local improvement into an exhaustive search method that enables finding an optimal solution *and* proving its optimality.

3 Nurse Scheduling as Hierarchical Constraint Satisfaction

The ConPlan project conducted at the German Research Center for Artificial Intelligence aimed at representing and solving nurse scheduling problems by generic techniques of partial constraint satisfaction. As a result a C++ library has been developed providing implementations of various search algorithms and constraint propagation techniques. This library is a part of the SIEDAplan nurse scheduling system that has been implemented by the SIEDA software house in Kaiserslautern and is currently used at the DRK hospital Neuwied. Working shifts are assigned to each nurse on each day of a certain period of time. A typical problem comprises 450 to 620 assignments that have to meet several requirements like

- legal regulations,
- optimized personnel costs,
- flexibility with respect to the actual expenditure of work,
- consideration of special qualities,
- management of vacation and absence,
- consideration of employee's requests,
- preference of working time models, that are common sequences of working shifts.

These requirements can be represented by constraints.

The problem has two characteristics:

1. It is hardly possible to fulfill all the requirements on a nurse schedule simultaneously. Conflicting requirements have to be distinguished due to their importance. While compliance with legal regulations is required, the consideration of employee's requests is optional. Additionally, explicit optimization tasks are part of the problem.

   Consequently, a solution is not necessarily consistent with all requirements but satisfies them as good as possible.

2. The satisfiability of requirements strongly depends on parameters like the contracts of the employees, the employees' working time balance, and the schedule of approved vacation. Hospitals are especially interested in opportunities to flexibly react on the current expenditure of work and to avoid expensive over-time work.

   Consequently, it is impossible to determine the most important and satisfiable requirements in advance.

Hence, the opportunity is needed to represent constraints of distinguished importance and to compute a solution complying with the set of most important constraints. Consequently, the nurse scheduling problem is represented as a *hierarchical constraint satisfaction problem (HCSP)*.

Common constraint logic programming approaches to nurse scheduling base on exhaustive search. None of these approaches manages soft constraints in order to represent optional requirements or optimization criteria.

3.1 Representation

To represent nurse scheduling as a HCSP one has at first to identify the constraint variables and their domains comprising the values the variable can be labeled with. In our representation, a constraint variable is generated for each nurse on each day that is considered by the schedule. Each variable has to be labeled with the shift the nurse has to serve at that particular day. At the moment, most hospitals still use a three-shift model with only one early-morning-shift `F1`, one day-turn `S1`, and one night-shift `N1`. Personnel scheduling is typically done by hand. Due to cost pressure and the deficiency of qualified and experienced personnel it has been recently recognized that working times must be much more flexible and efficient. A reasonable and promising solution seems to be the introduction of additional overlapping shifts (e.g. six- or nine-shift model) with less working hours. Hence, the system is required to integrate new types of shifts flexibly. These new shifts can be scheduled in a way, that the overlapping hours occur during very work-intensive periods. Additionally, some kinds of idle shifts

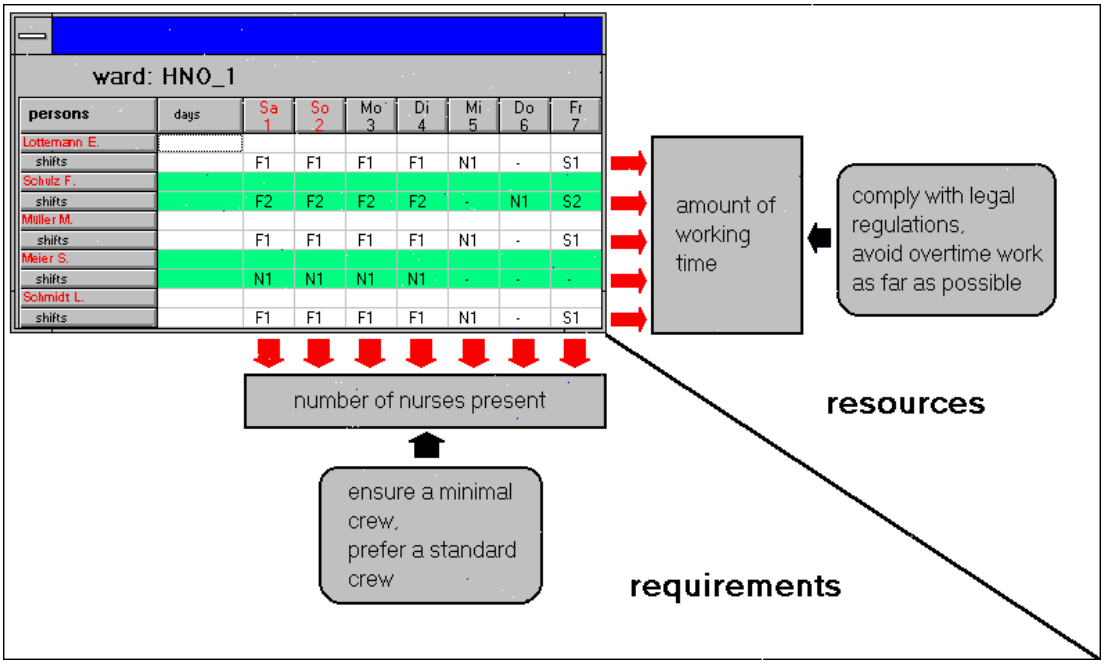


Figure 6: *Schema of the requirements in nurse scheduling.*

(— and *) as well as holidays UL have to be scheduled. Each of these particular shifts differ in start or end time, effect on the working time balance and costs. For a ward with a crew of 15 to 25 nurses and a planning period of one month this representation is a constraint problem of 450 to 620 variables each having a domain of around 10 or more shifts. About 10^{600} schedules are *a priori* possible solutions of the problem. Figure 6 gives an impression of the representation showing the cut of a schedule.

Constraints: These variables are connected by constraints representing demands on the schedule. These constraints are also shown in figure 6. The variables in each row of the schedule are connected by constraints concerning the number of nurses to be present at the ward at a particular time. This number can vary over the time, and both — a minimal and a preferred attendance of crew members at the ward — can be specified. The preferred size of a crew should not be exceeded. Contrarily, the variables in each column are connected by constraints concerning a particular employee. The most important constraints of this group involve the variables of a complete column and enforce a balance between the working hours to be served due to the employment contract and the scheduled working time. However, over-time work cannot always be avoided completely. The less over-time work is needed to achieve an acceptable crew attendance the better is the schedule. Additional constraints affect variables in a column of a schedule and concern obligational and preferred breaks between consecutive shifts and working time models — some preferred sequences of shifts defined for each employee. Working time models are traditionally used by the personnel department to control the deployment of particular employees. The schedule of each employee should be as similar as possible to one of its working time models. Constraints of the form "require the shift of employee *person* on day *day* to be equal to UL (abbreviated by *"shiftOfAt(person, day)* = UL") represent approved holidays. Employee's requests are translated into constraints like *shiftOfAt(person, day) is requested to end before 6 PM."*

Preference: Compulsory and optional demands on a schedule have been distinguished. Of course, the optional demands are of different importance. As mentioned above, two basic ways have been suggested to represent such different degrees of preference: Weights and priorities. The satisfaction of a constraint with a large weight can be less preferred than the simultaneous satisfaction of many constraints with smaller weights, whereas a constraint of high priority is always more important than all constraints of lower priority together. Both effects are needed in the nurse scheduling domain:

1. For instance cost reduction is always more important than respecting employee's requests. Thus, keeping the working time in balance has a higher priority than employee's requests.
2. In contrast, the schedule should respect *as many* requests of employees as possible. Thus, requests have an additional weight.

This problem can be solved distinguishing the following hierarchies of preference:

hard: compliance with legal regulations, and hard working time restrictions,

1: guarantee minimal crew,

2: management of working time, e.g. reduction of overtime work,

2: deploy a crew of preferred standard size,

3: compatibility of consecutive shifts, working time models,

4: consideration of employee's requests.

To refine this hierarchy each constraint c is mapped to a weight $\omega(c)$ to determine its preference in its hierarchy level. Two schedules are compared respecting the resulting hierarchy due to the following procedure:

0: If a schedule violates compulsory conditions then it is unacceptable. Nothing has to be compared.

1: In this hierarchy level, the weights are all 1.0. If one of the schedules ensures the minimal crew at more days than the other schedule, then it is preferred. Otherwise, the next hierarchy level will be considered.

2: In this level, the weight of each constraint is related to the distance between the number of working hours an employee is required to serve and the number of scheduled working hours. If the sum of these distances is equal in both schedules the next level will be considered.

3: Each optional condition on sequences of shifts served by the same nurse is weighted by 1.0. The more conditions are fulfilled, the better is the schedule. Requests of employees will be considered if the number of violated conditions is equal in both schedules.

4: Employees have the opportunity to state a certain number of heavier weighted requests and an arbitrary number of requests with normal weights.

As the rating of schedules is now defined, the next section will deal with the problem of how to construct good schedules to a given hierarchy of demands.

3.2 The Search Procedure

Typically, *branch&bound*, arc-consistency processing, and numerous application specific heuristics are coupled to achieve an acceptable latency of the system on *most* problem instances. Latency possibly varies strongly between different instances of the problem because of tree search's exponential complexity. Usually, the constraint representation of the scheduling problem has to be enriched by some heuristic constraints, which prune symmetric branches in the search tree, and manually programmed value selection strategies. Unfortunately, these heuristic control strategies interfere with the original constraint model. As a consequence, the relation between the original problem specification and the implemented constraint problem becomes fuzzy.

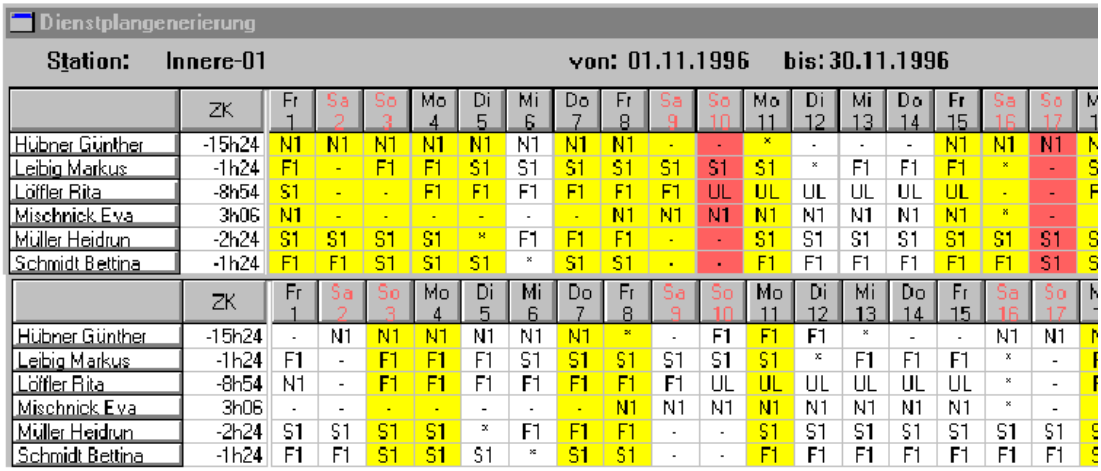
In contrast, iterative improvement algorithms due to Figure 5 have the advantage of being able to return a solution at any time — the result is of course not necessarily of a reasonable quality. Nevertheless, it is known that spending more time on searching improves the merit of the returned solution with respect to the original problem. Additionally, these algorithms are especially appropriate if some dialog with users is desired e.g. to request some hints for solving the problem. Hence, SIEDAplan conducts iterative improvement steps as presented in Figure 5.

However, Figure 5 leaves the question of how to find *bad* regions in a schedule open. The next paragraphs present a heuristic implementation for the procedure *choose-bad-region* that achieves an acceptable performance in the SIEDAplan system.

3.2.1 Finding Regions Where to Improve the Schedule

The problem of improving a schedule is reduced to the problem of searching for bad regions in a given schedule. The constraint graph and the set of violated constraints δ can be used to guide this search. In the SIEDAplan system the violated constraints are considered one by one for finding a promising region. Candidate variables for the set V' are computed according to the currently selected violated constraint and some heuristics that are partially described by the example below. The algorithm tries at first to repair the current schedule changing a single assignment in order to converge quickly on a sufficient schedule. Thus, a single variable is taken from this set of candidates at random as a unique element of V'. If the schedule has been improved by the following optimization step (cf. Figure 5, row 4), the next violated constraint will be chosen to improve the schedule. But of course this first try often fails. Two reasons are possible: On the one hand, changing the label of a single candidate may be possible but the procedure has chosen a wrong one. On the other hand, the procedure may have reached a local minimum that requires to change more than one label in order to achieve an improvement. Hence, the algorithm now stores *two* randomly chosen candidates in V' to improve the schedule in the next loop of the local search procedure. In the first case, the probability of choosing the right labels to change is increased. In the latter case, there is a chance to escape from the local minimum. If this try fails again, *three* variables will be chosen and so on until V' is of cardinality 8. Experience showed that optimizing the labeling of 9 or more variables is not worth its effort due to the current performance of the optimization library. Hence, if this bound is exceeded, the next constraint is chosen to compute a new set of candidates. This method enables the iterative improvement algorithm to repair easy deficiencies with small effort without getting stuck to local minima that could be improved by a human expert.

The best way to describe the nature of the applied heuristics more detailed is to consider a small example. Figure 7 shows two cuts of a schedule. The upper one represents the initial labeling of the variables by shifts. Forward checking of con-



Dienstplangenerierung

Station: Innere-01 von: 01.11.1996 bis: 30.11.1996

	ZK	Fr 1	Sa 2	So 3	Mo 4	Di 5	Mi 6	Do 7	Fr 8	Sa 9	So 10	Mo 11	Di 12	Mi 13	Do 14	Fr 15	Sa 16	So 17	M 1
Hübner Günther	-15h24	N1	N1	N1	N1	N1	N1	N1	N1	-	-	*	-	-	-	N1	N1	N1	N
Leibig Markus	-1h24	F1	-	F1	F1	S1	S1	S1	S1	S1	S1	S1	*	F1	F1	F1	*	-	S
Löffler Rita	-8h54	S1	-	-	F1	F1	F1	F1	F1	F1	UL	UL	UL	UL	UL	UL	-	-	F
Mischnick Eva	3h06	N1	-	-	-	-	-	-	N1	N1	N1	N1	N1	N1	N1	N1	*	-	
Müller Heidrun	-2h24	S1	S1	S1	S1	*	F1	F1	F1	-	-	S1	S1	S1	S1	S1	S1	S1	S
Schmidt Bettina	-1h24	F1	F1	S1	S1	S1	*	S1	S1	-	-	F1	F1	F1	F1	F1	F1	S1	S

	ZK	Fr 1	Sa 2	So 3	Mo 4	Di 5	Mi 6	Do 7	Fr 8	Sa 9	So 10	Mo 11	Di 12	Mi 13	Do 14	Fr 15	Sa 16	So 17	M 1
Hübner Günther	-15h24	-	N1	N1	N1	N1	N1	N1	*	-	F1	F1	F1	*	-	-	N1	N1	N
Leibig Markus	-1h24	F1	-	F1	F1	F1	S1	S1	S1	S1	S1	S1	*	F1	F1	F1	*	-	F
Löffler Rita	-8h54	N1	-	F1	F1	F1	F1	F1	F1	F1	UL	UL	UL	UL	UL	UL	*	-	F
Mischnick Eva	3h06	-	-	-	-	-	-	-	N1	N1	N1	N1	N1	N1	N1	N1	*	-	
Müller Heidrun	-2h24	S1	S1	S1	S1	*	F1	F1	F1	-	-	S1	S1	S1	S1	S1	S1	S1	S
Schmidt Bettina	-1h24	F1	F1	S1	S1	S1	*	S1	S1	-	-	F1	F1	F1	F1	F1	F1	F1	S

Figure 7: *Initial and improved labeling of the days with shifts.*

straints suffices to compute a schedule of this kind. Bad regions of the schedule have been shaded. Conspicuously many constraints concerning crew attendance are violated. These constraints have been neglected computing the initial labeling because the deployed heuristics of iterative improvement are especially appropriate to achieve the satisfaction of these constraints. Dark shaded rows represent the violation of a constraint demanding a minimal crew on the ward, light shadings denote a difference from the preferred attendance. For example, on Sunday the 10th nobody is at the ward during the early-morning shift F1. On Sunday the 17th two nurses attend during the day-time S1 but nobody serves an early-morning shift. On some days too many nurses are working, e.g. on Friday 1st, and Sunday 3rd. Note, that shifts on weekends have to be compensated by an idle shift (*) as early as possible.

The labelings to be improved (V') are chosen according to violated constraints. Some of these failures can be repaired very easily like for instance the constraint on crew attendance on Sunday 17th. As the improved schedule in the bottom of Figure 7 shows, only the change of a single label in variable

shiftOfAt(Schmidt, 17th)

is necessary to satisfy this constraint. Hence, an optimization of the labels directly affected by the violated constraints suffice to repair this constraint violation.

In contrast, the preferred crew attendance on Sunday 10th requires a more elaborate heuristic. On this day, too few employees are present at the ward. Hence, the algorithm chooses the variable concerning an employee not working on this day (in this case *shiftOfAt(Hübner, 10th)*) together with a variable in the same row at a day, when too many nurses are present (*shiftOfAt(Hübner, 8th)*) for V'. The *branch&bound* is called with this input to fill in better shifts.

These heuristics consider the available working time to be a more or less fixed resource. A request for more working time on a certain day has to be compensated at another day. By the way, this heuristic is very similar to the treatment of resources in resource-oriented configuration [9], the only local optimization strategy used in the field of knowledge-based configuration systems.

The schedule in the bottom of Figure 7 is the finally returned solution of the scheduling process. Even this small example of only five employees shows that generally not all demands on the schedule can be fulfilled. On some days more employees work than required because the available resources are typically not fully compatible with the

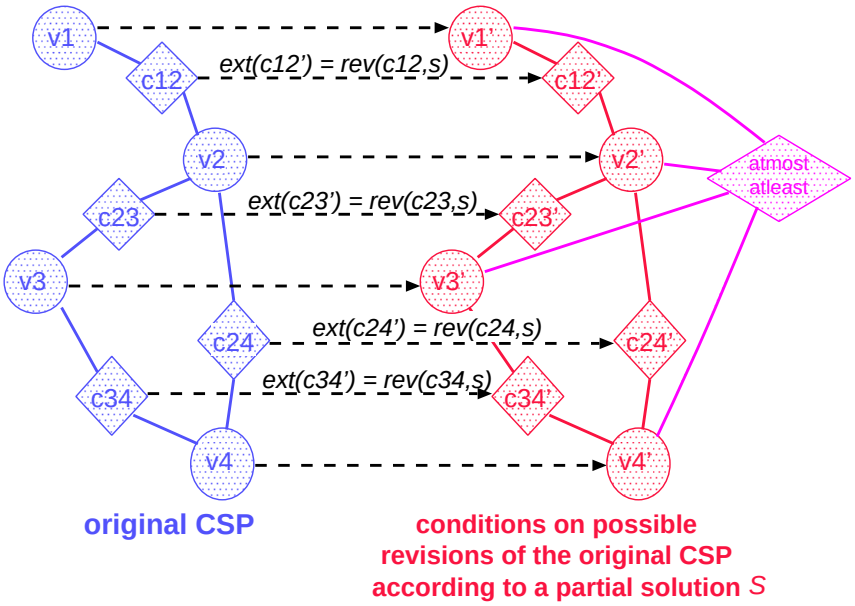


Figure 8: *Constraints on promising regions for changing the partial solution S of a CSP.*

expenditure of work. This fact has been the reason for representing nurse scheduling as a constraint optimization problem.

3.2.2 Experiences

The SIEDAplan system is able to compute a schedule for a ward of 15 to 20 nurses of reasonable quality within 5 to 20 Minutes on a PENTIUM 120 machine. These schedules comprise 450 to 620 constraint variables. However, the performance concerning both, run time and solution quality, depends strongly on the adequacy of the employed heuristics. The described application specific search method behaves significantly better than the standard algorithms for local search. In most cases, it is not possible to compute acceptable schedules by use of the *branch&bound*.

Mostly, compensation of large working time credits respectively debts is responsible for worse performance. However, these situations are difficult also for human experts which need hours instead of minutes to generate a schedule.

Performance of the *branch&bound* optimization steps becomes perceptibly better with growing quality of the improved schedule. Hence, it is a good strategy to reuse the set of violated constraints as an initial bound for the local optimization steps.

In order to overcome with unsatisfying results caused by inadequate problem specification or lack of adequate optimization heuristics, qualified personnel is allowed to include or remove constraints manually. Typically, soft constraints are added which prescribe a certain shift for a certain nurse on a certain day. If, for example, a nurse becomes ill in the planning period, this opportunity is also used for replanning while executing the schedule. The local search procedure can easily cope with removed or added constraints. Each change in the schedule is justified by increased quality. Hence, the system provides a sufficient degree of stability for interactive improvement of a schedule.

4 A Generic Method for Finding Bad Regions

The success of the heuristic procedure in the SIEDAplan system suggests to look for generic versions of this method. Methods of general applicability enable direct implementation of constraint models without need of representing control knowledge which is specific to the current problem.

A naive method for avoiding the application of search heuristics is the enumeration of *all possible* regions by the *choose-bad-region* procedure called in *iterative-improvement* (cf. Figure 5). When called by the overall search algorithm, this procedure first returns all sets of only one variable. If all of these variable sets have been enumerated, then all sets of two variables are returned and so on. Finally, after trying to improve $V' = V$, one knows that a global solution has been computed. However, this method neglects all information that can be retrieved from the constraint graph and the deficiencies of the current solution.

Our idea on a generic procedure for finding bad regions in partial solutions of a CSP now is to constrain this naive enumeration of regions for solution improvement by the available information. In the following, promising regions for solution improvement are called to form global revision sets. The problem of finding global revisions is formulated as a boolean CSP whose constraint graph is very similar to the original problem (cf. Figure 8). If a solution of the boolean problem assigns a 1 to a variable, the corresponding variable in the original problem is considered to be a part of a promising region for improvement. Each of the constraints in this boolean problem represents a necessary condition on promising regions that only depend on *one* constraint in the original problem and a partial solution S. In the following, the regions complying with the condition concerning a single constraint are called to define a set of local revisions. The constraints *atmost* and *atleast* are well known from scheduling systems and count here the occurrences of 1 in the solution of the binary problem. These constraints can be used to control the size of the regions. Hence, it is possible to return small revisions first in order to conduct cheap optimization steps first.

4.1 Local Revision Sets

Firstly, the relation of the original problem to the abstract problem of finding regions for local repair according to Figure 8 needs to be defined. In the following, v' always denotes the variable in the abstract boolean problem that corresponds to variable v in the original problem. Analogously, c' represents the constraint in the abstract problem referring to constraint c in the original problem. The whole set of variables in the abstract problem is written as V_{rev}, the set of constraints as C_{rev}.

Definition 7 *Let l_1 and l_2 be labelings of all variables in V. Then* $\mathrm{diff}(l_1,\ l_2)$ *returns a labeling of V_{rev} according to the following definition.*

$$\forall v' \in V_{rev} : \mathrm{diff}(l_1,\ l_2)\downarrow_{v'} = \begin{cases} 0 & \textit{iff } l_1\downarrow_v = l_2\downarrow_v \\ 1 & \textit{otherwise} \end{cases}$$

The purpose of the *diff*–function is to deal with differences between labelings in boolean constraint problems.

Definition 8 *Local revisions: Let l be a labeling of the variables in V and c be a constraint in the original problem. Then, the smallest local revision set of l respecting c is defined as follows:*

$$\mathrm{rev}(c,\ l) := \bigcup_{l' \in \mathrm{ext}(c)} \{\mathrm{diff}(l\downarrow_{V(c)},\ l')\}.$$

All extensions comprising rev(c, l) *are called local revision sets of* l *respecting* c.

Local revision sets cover all differences of a current labeling l to all labelings complying with constraint c. Although defined extensionally here, intensional definitions for common constraints, which are typically defined by propagation methods in a constraint library, are possible.

Intensional definitions of local revisions will usually approximate the smallest local revision set by supersets which can be propagated efficiently. The basic idea is to forbid as many tuples in the revision set as possible which represent futile steps of repair. As an example, consider the constraint from the SIEDAplan application (cf. section 3) that enforces the working time of an employee not to exceed an upper bound of *max* minutes:

$$resource_{f,max}(x_1, \ldots, x_n) \equiv f(x_1, \ldots, x_n) \leq max, x_1 \in D_1, \ldots, x_n \in D_n.$$

As stated in section 3.1, the variables x_1 to x_n are labeled with shifts to be served by the same employee on consecutive days. Function f returns the working time in minutes corresponding to the currently assigned working shift (e.g. *f(*F1*) = 450, f(—) = 0*). In this example, the domains D_1 to D_n comprise the possible shifts (e.g. F1, S1, N1, UL, *, and —). The following procedure illustrates an approximative propagation method for rev(*resource*, l) that is efficient.

- generate a labeling l_{min} with $f(l_{min} \downarrow_{x_i})$ is minimal for all $i \in 1, \ldots, n$.
- set $ext := \{0, 1\}^n$,
- for each $i \in \{1, \ldots, n\}$ do
  - if labeling $(l_{min} \setminus \{l_{min} \downarrow_{x_i}\}) \cup \{l \downarrow_{x_i}\}$ violates constraint *resource* then remove all labelings from ext which label x_i' with 0,
  - else for each $j \in \{1 \ldots, i-1, i+1, \ldots, n\}$ do
    - if labeling $(l_{min} \setminus \{l_{min} \downarrow_{x_i}, l_{min} \downarrow_{x_j}\}) \cup \{l \downarrow_{x_i}, l \downarrow_{x_j}\}$ violates constraint *resource* then remove all labelings from ext which label x_i' and x_j' with 0,
- return ext as the extension of rev(*resource*, l).

For each pair of labels of two variables x_i and x_j, this procedure finds out whether the constraint can be satisfied obtaining these labels[2]. If it is impossible to satisfy the constraint then all repair steps with $x_i = 0$ and $x_j = 0$ are forbidden. What is the effect of propagating this revision? Assume for instance that too much working time has been scheduled for a person p — constraint *resource* is violated. In this case, rev(*resource*, l) will suggest only changes in a region where p works and less work is possible. If otherwise a change is required on a day d when p serves an idle shift then rev(*resource*, l) enforces supporting changes in order to compensate the additionally required working time on day d. Consequently, this revision strategy implements the main part of the heuristic procedure for finding regions of repair in a nurse schedule (cf. section 3.2.1).

The local revision set of a constraint depends in at least one point on the labeling l. The revision set only comprises a 0-tuple if l satisfies c. Otherwise, at least one assignment has to be changed.

[2]Of course, an analogous procedure can be additionally conducted for any group of three and four labels etc. This is simply a question of the effort one is allowed to spend on the propagation of the revision.

Property 3 *Let*

$$rev_{anychange}(c,l) = \begin{cases} \{0,1\}^{|V(c)|} & \textit{iff } l \textit{ satisfies } c \\ \{0,1\}^{|V(c)|} \setminus \{(0,\ldots,0)\} & \textit{iff } l \textit{ violates } c \end{cases}$$

Then, $rev_{anychange}$ is a local revision of c.

Proof: $\mathrm{rev}_{anychange}$ contains nearly the complete cartesian product $\{0,1\}^{|V(c)|}$ except in one case: If c is violated then a change is required. Trivially, for any $l' \in \mathrm{ext}(c)$ property $\mathrm{diff}(l\downarrow_{V(c)},\ l') \neq (0,\ldots,0)$ holds true. Consequently,

$$\mathrm{rev}_{anychange}(c,l) \subseteq \bigcup_{l'\in \mathrm{ext}(c)} \{\mathrm{diff}(l\downarrow_{V(c)},\ l')\}.$$

♢

Note: Property 3 shows an easy way to construct default local revisions for any constraint in a constraint library. However, this easy method will usually provide only poor approximations of the smallest local revision.

4.2 Global Revision Sets

Global revisions consider sets of constraints instead of single constraints.

Definition 9 *The* smallest *revision set of a partial solution l respecting constraint set C' is defined as*

$$\mathrm{rev}(C',\ l) := \bigcup_{l'\in \mathrm{ext}(C')} \{\mathrm{diff}(l,\ l')\}.$$

All extensions comprising $\mathrm{rev}(C',\ l)$ *are called revision sets of l respecting C'. Revision sets respecting all constraints in C are called global revision sets.*

Property 4

$$\bowtie_{c\in C'} \mathrm{rev}(c,\ l) \supseteq \mathrm{rev}(C',\ l).$$

Obviously, the join of local revisions above denotes exactly the set of all solutions to the abstract problem in Figure 8. Hence, the theorem claims that the solutions of this abstract constraint problem form a global revision set of labeling l. However, this global revision set is generally not minimal.

Proof: One can proof by a few equations that for any $l' \in \mathrm{ext}(C')$ the difference to the current label $\mathrm{diff}(l',\ l)$ is in $\bowtie_{c\in C'} \mathrm{rev}(c,\ l)$ presupposing that all variables in l are directly affected by a constraint in C'

$$\begin{aligned} \bowtie_{c\in C'} \mathrm{rev}(c,\ l) &= \bowtie_{c\in C'} (\{\mathrm{diff}(l'\downarrow_{V(c)},\ l\downarrow_{V(c)})\} \cup \mathrm{rev}(c,\ l)) \\ &= \{\mathrm{diff}(l',\ l)\} \cup \bowtie_{c\in C'} \mathrm{rev}(c,\ l). \end{aligned}$$

The first equation follows from the definition of local revision sets. The second presupposes that all variables in V are directly affected by the constraints in C' and

$$\mathrm{diff}(l'\downarrow_{V(c)},\ l\downarrow_{V(c)}) \in \mathrm{rev}(c,\ l).$$

♢

choose-bad-region(V, C, $\succ$, l, δ, variant $\in$ {by_size, by_hierarchy})

1. *if COP_{rev} has no value (this function is called for the first time) or δ has been improved by the last optimization step in* iterative-improvement *then begin*
   (a) $COP_{rev} := (V_{rev}, C_{rev}, \{0,1\}, \succ')$ *with* $C_{rev} = \bigcup_{c \in C} \mathrm{rev}(c,\ l)$ *and* $C_1 \succ' C_2 \iff \{c \mid c' \in C_1\} \succ \{c \mid c' \in C_2\}$;
   (b) $n := 1$; $level = 1$;
   (c) *add the following constraint to* COP_{rev}*:* atleast *and* atmost *n occurrence of 1*

   end;
2. *if variant = by_hierarchy*

   then begin
   (a) $\delta_{relevant} := \delta \cap \bigcup_{i=1}^{level} C_i$;
   (b) $\delta_{rev} := \bigcup_{c \in \delta_{relevant}} \mathrm{rev}(c,\ l)$
   end

   else $\delta_{rev} := \bigcup_{c \in \delta} \mathrm{rev}(c,\ l)$;
3. *call* branch&bound *(enhanced by forward checking etc.) with δ_{rev} as bound on COP_{rev} for the* next *partial solution better than δ_{rev};*
4. *if a partial solution l has been found return* $V' = \{v \mid l \downarrow_{v'} = 1\}$;
5. *if variant = by_hierarchy and there is a hierarchy level C_i with $i > level$ then begin*
   (a) $level := level + 1$;
   (b) *goto row 2*

   end;
6. *if no partial solution is available and* $n \leq \mid V \mid$ *then do begin*
   (a) $n := n + 1$;
   (b) *reset COP_{rev} and remove* atleast *and* atmost *constraints;*
   (c) *add the following constraints to COP_{rev}:* atleast *and* atmost *n occurrence of 1;*
   (d) *goto row 3*

   end;
7. *l is optimal. Hence, return* $V' = \{\}$.

Figure 9: *Enumerating revisions of optimization problems.*

Note: Property 4 also holds for the most important set of constraints which is satisfied by an optimal solution in an optimization problem. Consequently, property 4 is applicable to COPs, as well.

4.3 Searching with Global Revision Sets

The basic idea for applying these results is to search the original problem by algorithm *iterative-improvement* according to Figure 5 controlled by an exhaustive search of the abstract problem that enumerates global revisions. The resulting hybrid algorithm still is an anytime-algorithm because partial solutions are available all the time. Additionally, proving optimality is possible due to the results of the previous section. If all global revisions have been searched without improving the current partial solution l, then l is optimal.

no.	den.	sat.	algo.	time/s			checks/10^6			assignments/10^3		
				⌀	min.	max	⌀	min.	max	⌀	min.	max
20 variables												
1	1.0	0.5	egr+MACall	1573	314	2980	56.0	88.4	130.2	52.4	9.6	115.4
2	1.0	0.5	bb+FC	1166	372	2339	35.5	11.9	68.8	31.1	11.8	60.3
3	1.0	0.7	egr+MACall	1118	294	3216	34.1	8.2	100.2	24.3	6.6	67.7
4	1.0	0.7	bb+FC	725	232	1446	24.6	7.7	49.1	17.2	5.6	33.7
5	0.5	0.5	egr+MACall	253	81	849	7.4	2.0	26.7	14.4	4.4	50.1
6	0.5	0.5	bb+FC	182	52	383	6.2	1.8	12.8	10.8	3.1	21.6
7	0.5	0.7	egr+MACall	17	11	22	0.3	0.2	0.4	0.7	0.4	0.8
8	0.5	0.7	bb+FC	56	8	173	2.1	0.3	6.6	2.5	0.4	8.5
30 variables												
9	0.44	0.7	egr+MACall	4019	704	11777	146	26	508	153	30	535
10	0.44	0.7	bb+FC	4348	28	12785	139	62	336	133	62	323
11	0.22	0.5	egr+MACall	755	208	2522	15.7	4.2	57.9	44	12	156
12	0.22	0.5	bb+FC	1047	547	2294	61.4	16.7	304.6	145	46	635

Figure 10: *First empirical comparison of* bb+FC *and enumeration of global revisions* (egr+MACall). *All the problems consist of 20 respectively 30 variables with a domain of 10 values. Constraints have been grouped into 6 hierarchy level of nearly equal size.*

Figure 9 describes the idea of two enumeration algorithms for constraint optimization problems: *by_size* and *by_hierarchy*. At first, let us consider variant *by_size* and neglect all if-branches concerning *by_hierarchy*.

COP_{rev} holds the constraint problem for finding global revisions. As mentioned above, *atmost* and *atleast* constraints are used to enumerate global revisions according to their size in order to conduct cheap optimization steps first in algorithm *iterative-improvement*. The variable n, occurring in the rows 1 and 6, controls the number of assignments to be retracted. In row 1, COP_{rev} is built up again after improving l because the constraints in C_{rev} typically depend on l. Variable δ_{rev}, which is set in row 2, serves as a bound in the following call of the *branch&bound* procedure. Only revisions promising to improve l are returned by the enumeration in row 4. If the *branch&bound* fails to find a new partial solution of COP_{rev} better than δ_{rev} then, occasionally, larger revisions are required (row 6). If no larger revisions are available then l is optimal. Search in *iterative-improvement* terminates because *choose-bad-region* returns an empty variable set.

Variant *by_hierarchy* works very similar although only applicable to constraint hierarchies. The only difference is that it tries to satisfy important hierarchy levels before levels of minor importance. Therefore, variable *level* is maintained in the algorithm starting with a value of 1. When computing the bound in row 2 for calling the *branch&bound* on the revision problem, variant *by_hierarchy* neglects all constraints of the hierarchy levels below *level*. Hence, the next global revision is required to promise an improvement within or above *level* — the currently violated constraints in more important hierarchy levels will be satisfied first.

5 Experiences on Solving Randomly Generated Problems

As mentioned above, the enumeration of global revisions is a structural method that performs best if it is possible to improve complete labelings by small changes. The major drawback of this method is the overhead which is caused by the effort of searching the abstract constraint problem.

Randomly generated problems exhibit no special structure. Thus, experiments on

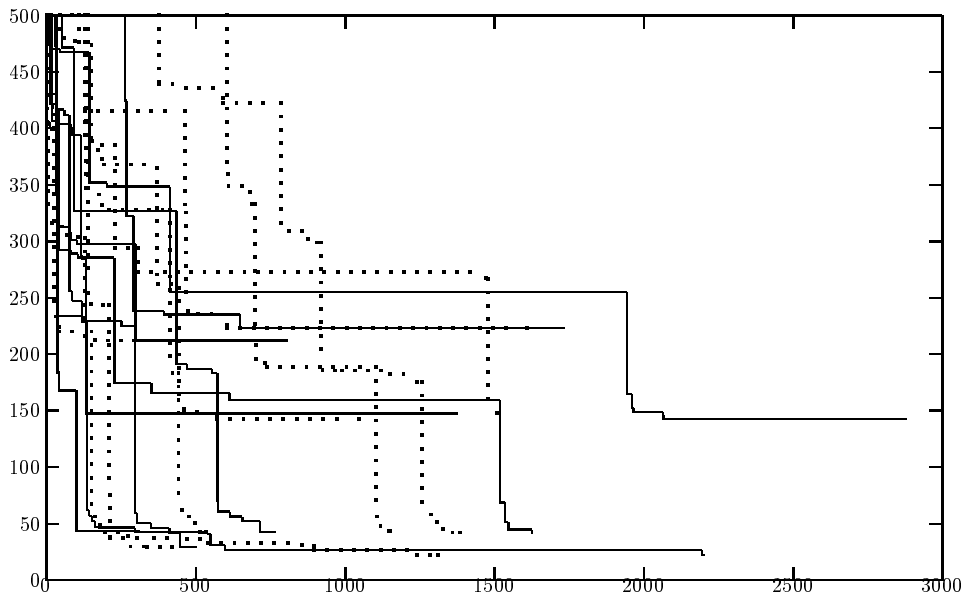


Figure 11: *Improvement of solution quality (in weight of violated constraints) over time (in seconds) — comparison between* bb+FC *(dotted curves) and* egr+MACall *(continous curves). The 10 problems all comprise 20 variables with a domain of 10 values. Density of the constraint graph is 1.0. The constraints of satisfiability 0.5 have been grouped into 6 hierarchy levels.*

these problems are appropriate to answer the question:

Is the effort of searching for global revisions acceptable?

Two kinds of experiments prove that the answer to this question is *"yes"*. Enumeration of global revisions and an enhanced *branch&bound* are compared in searching exhaustively. Secondly, local search algorithms and *branch&bound* variants are rated according to their performance on larger problems that are not searched exhaustively.

Usually, randomly generated binary HCSPs are generated with respect to the following properties

Number of variables: the main measure for the problem size,

domain size: all variables have the same domain,

density: the ratio of the number of generated constraints to the number of constraints in a completely connected constraint graph,

satisfiability: the ratio of the number of tuples in the extension of the constraints to the size of the cartesian product of two domains,

number of hierarchy levels: the constraints are grouped into a number of hierarchy levels of nearly equal size— this property is of course specific to HCSPs.

Within each hierarchy level the constraints are rated according to a random weight.

5.1 Exhaustive Search

Figure 10 presents results on experiments over 60 randomly generated problems of 20 and 30 variables with a domain of 10 values on a SUN ULTRA 2 machine. Both procedures, *branch&bound (bb+FC)* search as well as *enumeration of global revisions (egr+MACall)*, can occur in many variations differing in the employed constraint processing techniques (cf. section 2.2.1). The *bb+FC* algorithm in Figure 10 uses e.g. the

maximal width and maximal constraint number as criteria for a static variable ordering. Additionally, variables are ordered dynamically by a *pessimistic minimum remaining value (MRVpess)* heuristic that exploits the hierarchical structure of the constraint problem. *Forward checking* and *backmarking* is done with hard and soft constraints. This is a relatively well performing variant of the *bb+FC* algorithm. The same procedure is used enumerating global revisions *(egr+MACall)* when searching the original problem. Searching the abstract problem, arc-consistent compatibilities are maintained in order to find more promising revisions early.

Figure 10 presents run time, number of checks, and the number of assignments in order to assess performance of the algorithms. Using the *egr+MACall* algorithm, checks and assignments have been counted in the original as well as in the abstract problem.

Except the rows 7 and 8 *egr+MACall* performs worse on the smaller problems according to this data. The rows 7 and 8 concern problems of a density of 0.5 and a constraint satisfiability of 0.7 where each of the generated instances has a solution satisfying *all* constraints. These problems are *not* over-constrained although represented with soft constraints. In contrast to the other problems it has not been necessary to prove optimality of a computed solution. Apparently, *egr+MACall* is superior in finding an optimal solution but inferior in proving optimality when working on problems of this size.

A closer look at the results of the other experiments confirms this assumption. Figure 11 displays the way how the quality of a solution has been improved over time in the experiments referring to the rows 1 and 2 in Figure 10. Each continuous curve refers to a run of the *egr+MACall* method, each dotted curve belongs to a run of the *bb+FC* algorithm. A point (x, y) on the curve states that at time x (in seconds) the best visited complete labeling violated constraints of weight y. Hierarchy levels have been translated into global weights according to property 2. On 7 of 10 problem instances *egr+MACall* has been able to find (nearly) optimal solutions significantly earlier than the *bb+FC* algorithm. However, proving optimality took always more time using *egr+MACall*. In the experiments referring to the rows 2 to 8 *egr+MACall* has always been faster in finding the optimal solution. In these experiments, proving optimality of a solution took always much more time than finding it.

The rows 9 to 12 show that *egr+MACall* is superior to *bb+FC* when solving larger problems. This effect is probably caused by *egr+MACall's* more flexible systematics in search. Obviously, the performance of pure tree search algorithms depends much more on the order in which the variables are labeled. In contrast, *egr+MACall* is able to change early assignments without searching large portions of the search space exhaustively. Hence, nearly optimal solutions are found earlier and can be used to prune the search space more effectively when proving optimality. The importance of this point is likely to grow with the size of the search problem.

5.2 Convergence On Good Solutions

The previous section showed that quick convergence on good solution is a major performance criterion. This section reports results from about 420 experiments on randomly generated problems of 40 variables and 280 experiments on problems of size 100. On problems of this size, exhaustive search for an optimal solution is usually not possible. Hence, the experiments stopped searching after 5 respectively 15 minutes.

The experiments cover all three groups of constraint algorithms that have been described above by the following instances:

1. Enumerating global revisions:

- egr+FC uses the same strategies searching the original as well as the abstract problem: *forward checking* of hard and soft constraints, maximal width and maximal constraint number as static variable ordering heuristic, and MRV pessimistic as dynamic variable ordering heuristic.
- egrh+FC uses the same search strategies as egr+FC but follows the by_hierarchy enumeration strategy. Hence, the performance of this variant shows whether enumeration strategies can benefit from the hierarchical structure of a constraint problem.
- egr+MACall deploys the strategies mentioned above searching the original problem. The abstract problem is searched maintaining compatibilities after each assignment and using MRV optimistic as dynamic variable ordering. On the one hand, this procedure spends more effort on finding global revisions. On the other hand, the increased level of consistency processing searching the abstract problem is another method for trying promising revisions first.

2. Branch&bound search:
   - bb+FC deploys the same search techniques as egr+FC. Experience showed that *forward checking* is the least amount of consistency processing to make *branch&bound* tractable.
   - In contrast, bb+MAXall computes consistent compatibilities after each assignment and uses MRV optimistic as dynamic variable ordering heuristic.
3. Local search: After computing an initial labeling using *forward checking* of hard and soft constraints, mincon and minconwalk are used as described above.

The Figures 12 to 16 show the results of the experiments. The x–coordinate represents run-time in seconds whereas the y–coordinate reports the average sum of constraint weights that are caused by constraint violations of the best labeling visited at that time. Each curve represents an average of the same 10 randomly generated example problems. Again, hierarchy levels have been translated into constraint weights by property 2.

On the performance of the *branch&bound* instances: The Figures 12 to 16 show the performance of the used *branch&bound* instances in column 2. Typically, *branch&bound* search leads to a rapid improvement within the first few seconds which is achieved by the first complete labeling of all variables. However, the algorithms are mostly not able to improve this first labeling significantly within the time bound because the algorithm is only able to change the labels of the variables which are near the leaves of the search tree. In this situation, the strategies of value assignment and variable ordering become essential. bb+MACall does in many cases a pretty good job in this area at the cost of additional costs for consistency processing — the first complete labeling is found significantly late. If the domains are not larger than 20 values, consistent compatibilities suffice to assign promising values first. In several cases, variables are ordered in such a way that significant improvements are possible changing only a few late assignments (cf. experiment 2 of Figure 16). However, as practical experience of the SIEDAplan project shows, a sufficient number of equally sized hierarchy levels seems to be a precondition of this good performance.

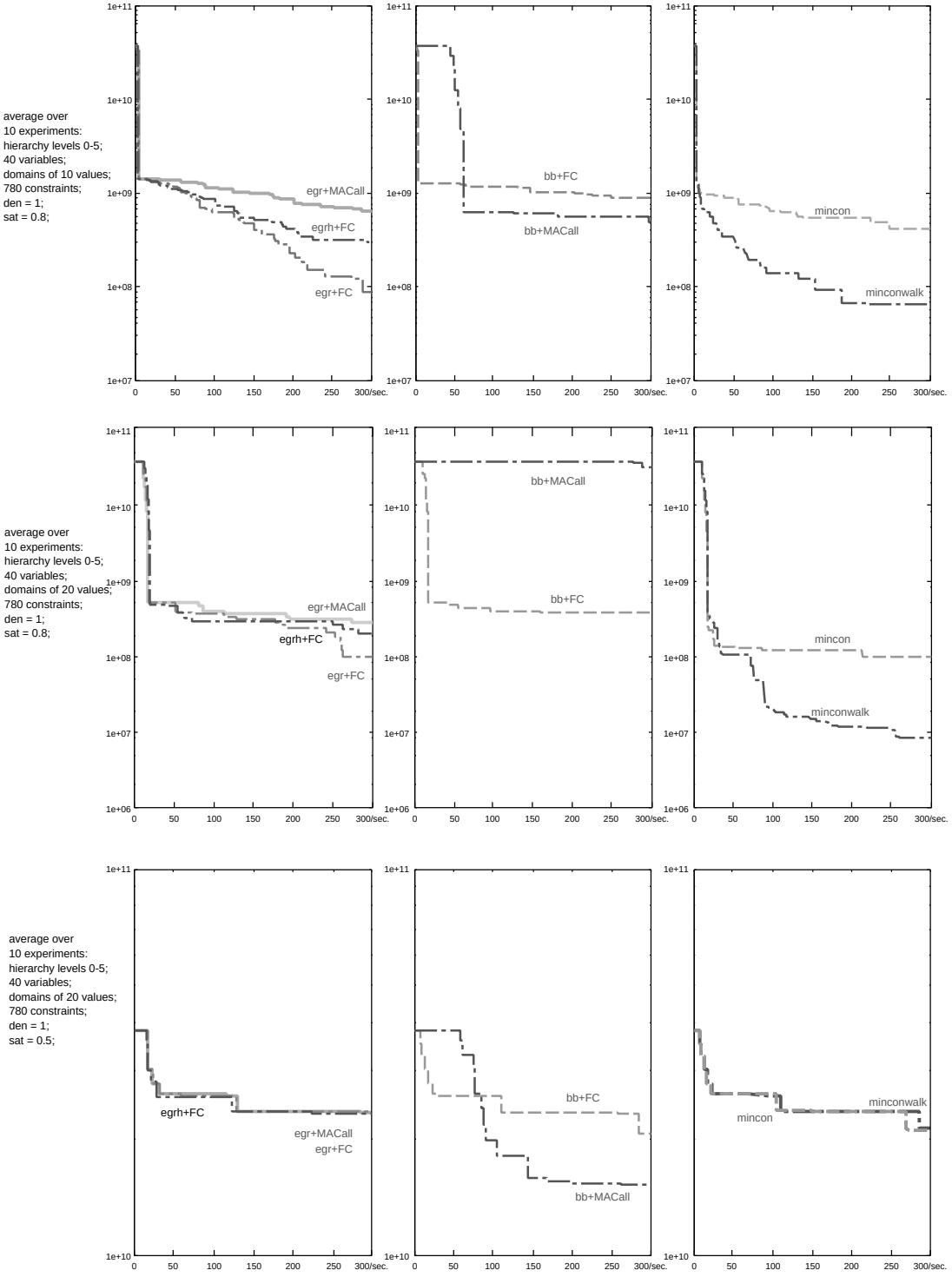


Figure 12: *Performance of different algorithms on high density constraint problems with 40 variables.*

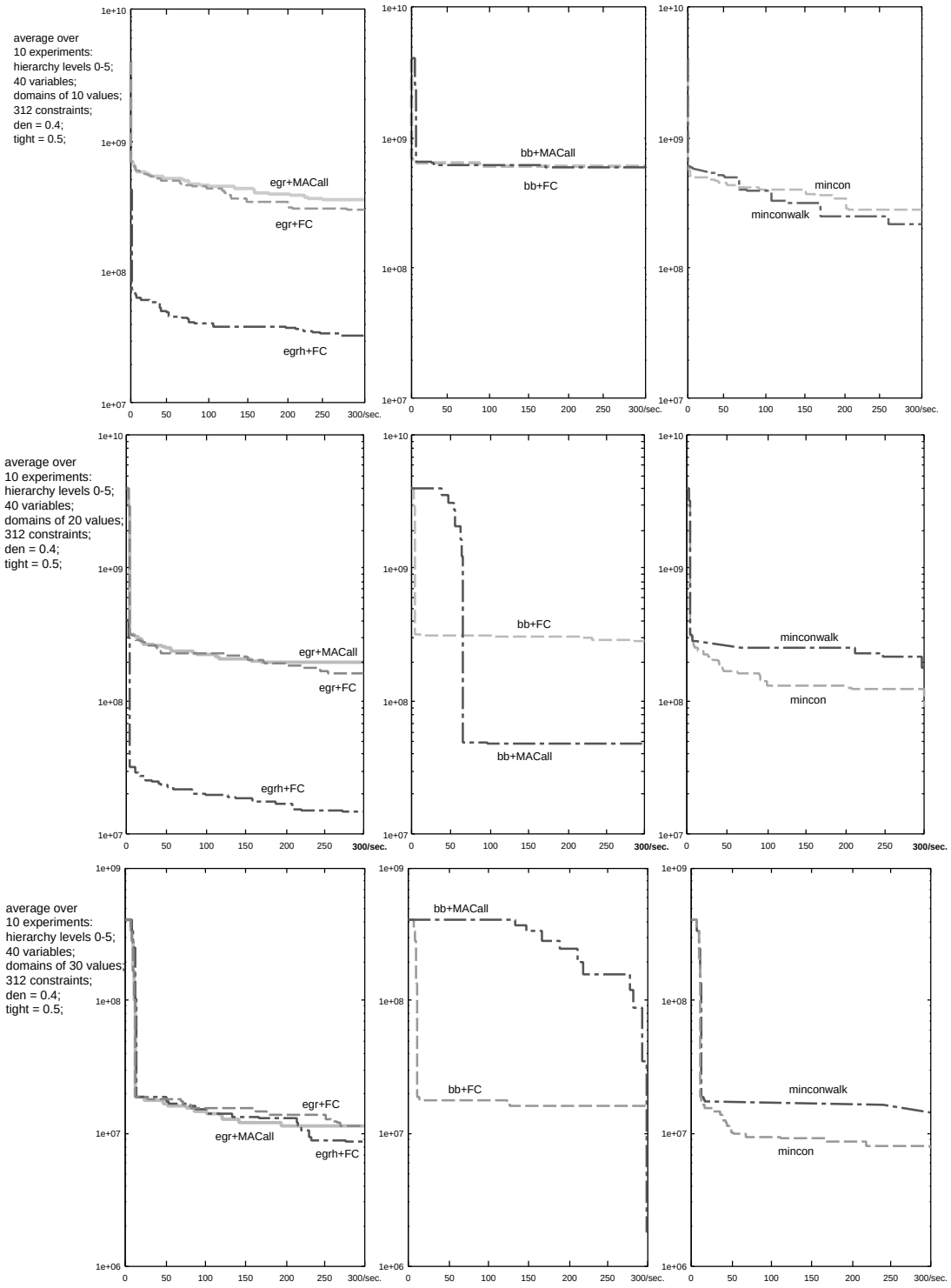


Figure 13: *Performance of different algorithms on constraint problems of medium density with 40 variables.*

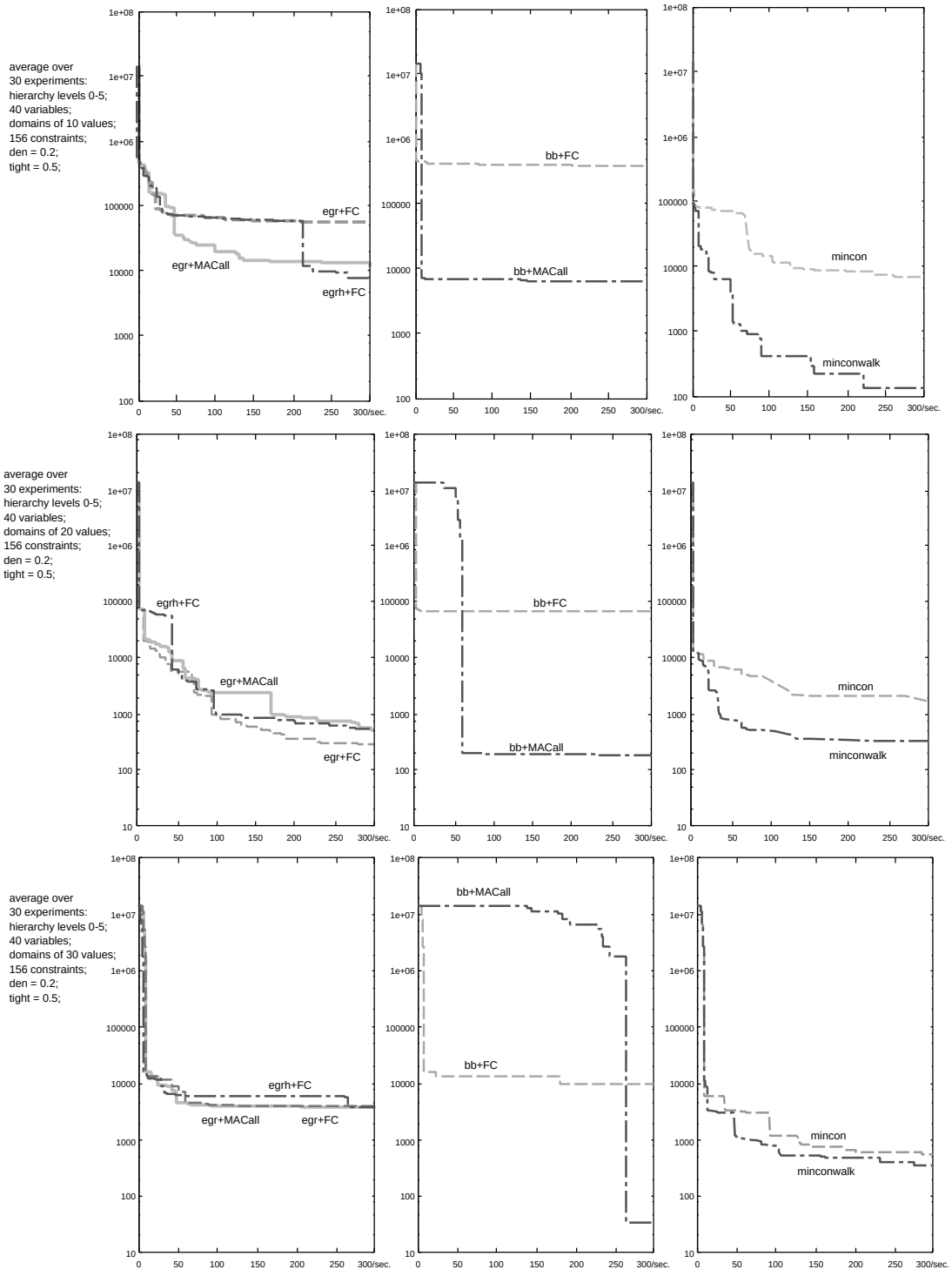


Figure 14: *Performance of different algorithms on lower density constraint problems with 40 variables.*

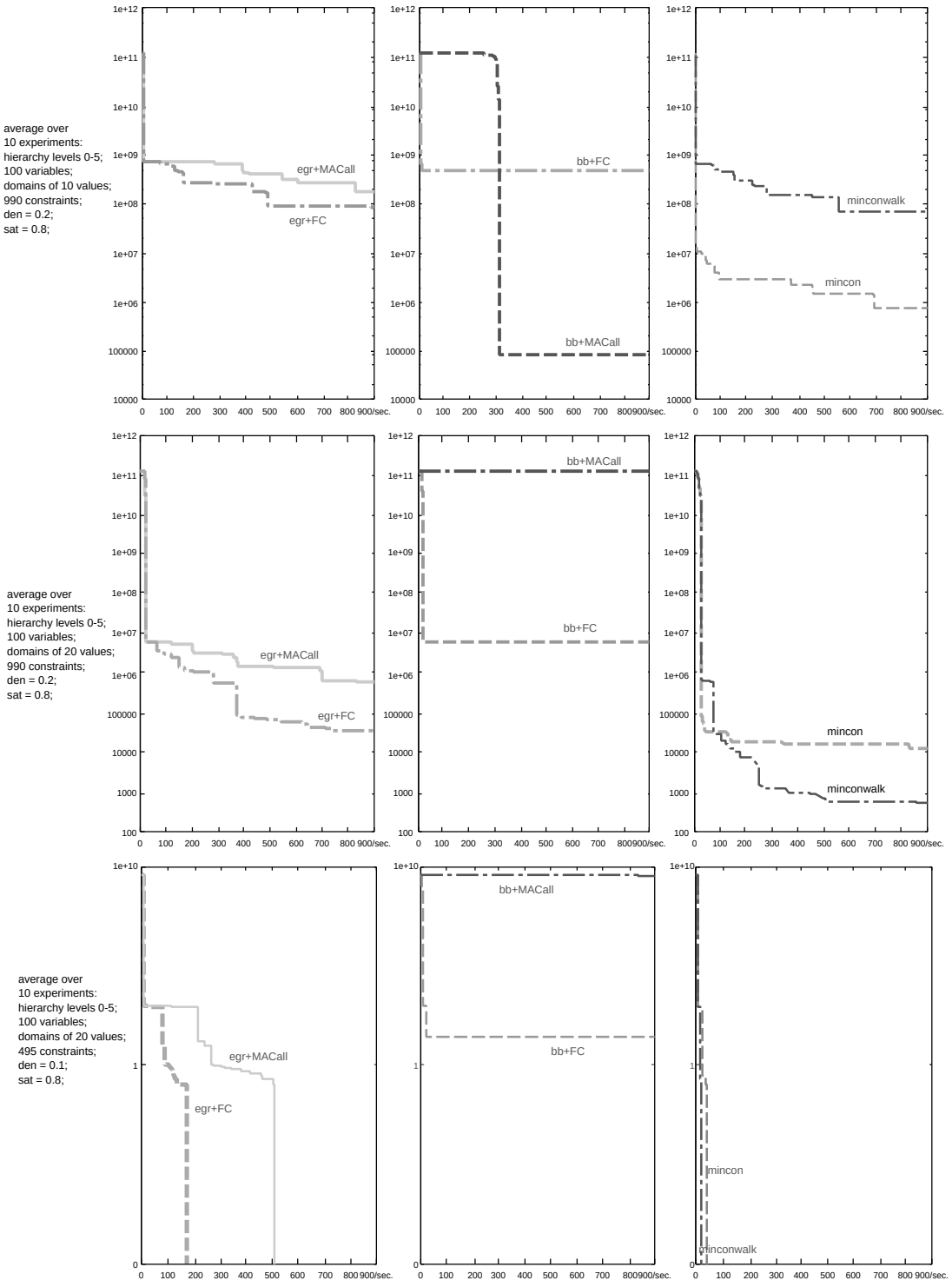


Figure 15: *Performance of different algorithms on highly satisfiable constraints — problem size of 100 variables.*

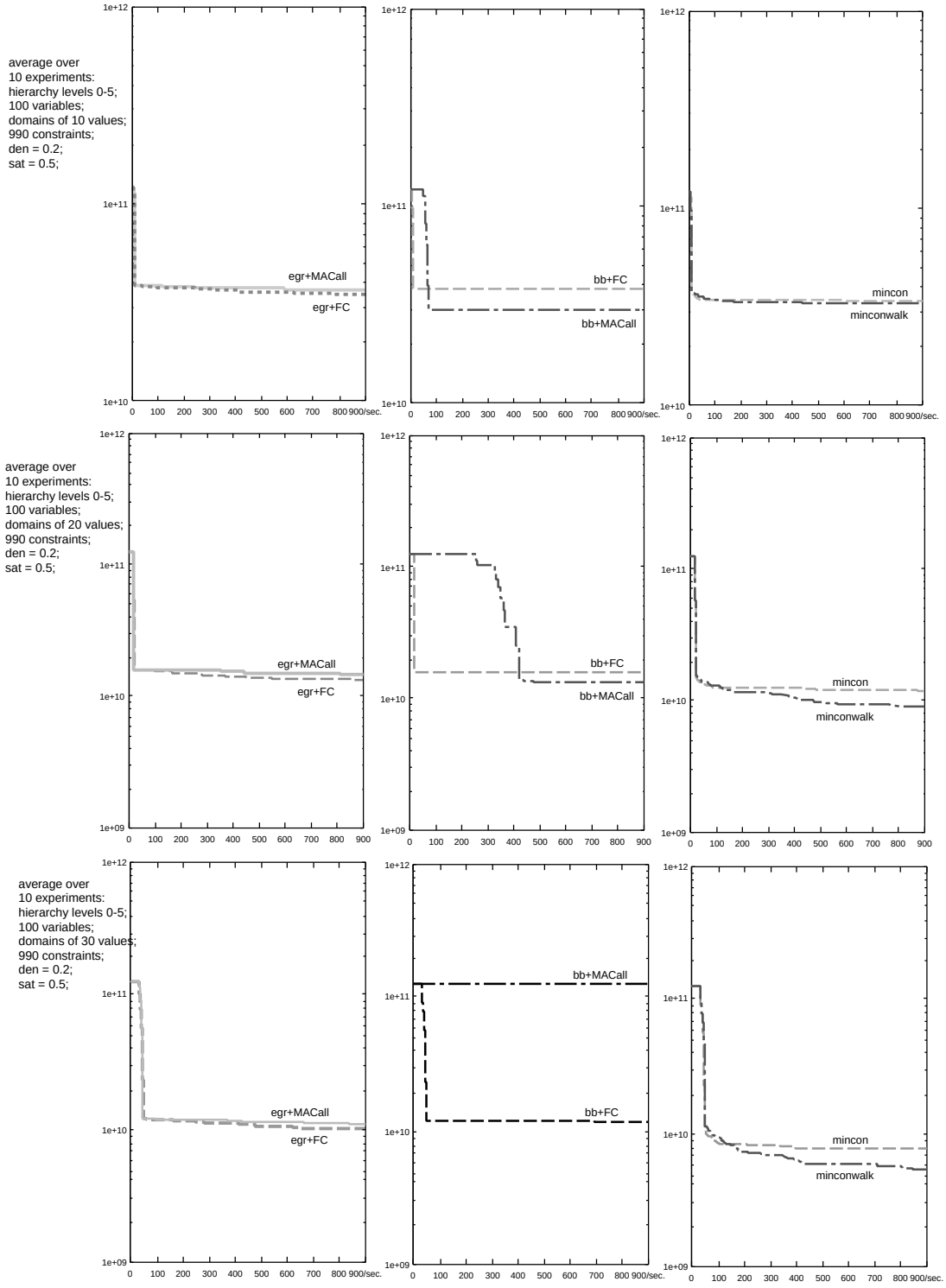


Figure 16: *Performance of different algorithms on constraints of medium satisfiability — problem size of 100 variables.*

The performance of local search: In the diagrams, results concerning the local search algorithms are presented in column 3. Mincon and minconwalk are the methods of choice when solving unstructured and larger problems. One can hardly determine in advance which of these algorithms performs better on a given task. Note that local search has been done on initial labelings which have been computed by use *forward checking* with hard and soft constraints.

Comparison with instances of *egr*: The performance of the *egr* instances is presented in row 1 of the Figures 12 to 16. All instances of *egr* have been better than the standard *branch&bound* instance which uses *forward checking* and *pessimistic MRV*. Hence, the overhead of searching global revisions is acceptable even when solving unstructured problems. Generally, it seems not to be a good idea to use a too large degree on prospective constraint processing in the abstract problem. On searching low density problems (refer to Figure 14), computation of consistent compatibilities in algorithm egr+MACall sometimes pays because apparently more important constraint revisions are tried out earlier. However, in all the other cases, this variant performed worse than the others.

On high density problems, egr+FC performed better than by_hierarchy enumeration. On these problems, egrh+FC apparently searches for revisions of high priority constraints which cannot be satisfied. However, on medium density problems, egrh+FC performed better than *all* the other algorithms. This encouraging results give rise to the assumption that this strategy of enumerating global revisions will have an impact on practice in constraint-based search.

In all experiments, the behaviour of *egr* was somehow in between the behaviour of pure tree search and local search. The *egr* algorithms inherit the ability of improving a solution constantly by small steps from local search. From *tree search* they inherit the guarantee of finding an optimal solution.

Beside all the differences in the performance of the described algorithms, Figure 12 and 16 show the following. Although following very different principles, the algorithms behave similar if density and tightness of the given constraint problem are sufficiently high, i.e. if the problem specification represents a lot of knowledge on the desired result. Hence, more knowledge may usually not imply less search but makes the control of search less important.

6 Conclusion

In this report, the standard algorithms for solving *constraint optimization problems* including hierarchical constraints have been compared to the scheme of *iterative improvement*. On the one hand, this framework makes use of latest progress in tree search on constraint optimization problems because it deploys the *branch&bound* algorithm including constraint processing extensions. On the other hand, algorithms of this framework exhibit a robust run time behaviour and they are able to cope with dynamically changing problem specifications. A commercial scheduling application of such an algorithm is presented in order to prove applicability of this scheme.

Additionally, the report presented generic algorithms for enumerating global revisions exhaustively in order to overcome with the two basic drawbacks of iterative repair algorithms:

1. Usually, problem specific control knowledge is required to find regions where the *branch&bound* is called to repair the current solution.

2. Heuristic algorithms for iterative repair are not able to search constraint problems exhaustively.

Studies on randomly generated problems proved that the overhead, which is caused for this enumeration procedure, does not affect applicability of this new generic search paradigm. On larger problem sizes, enumerating global revisions outperforms *branch& bound* algorithms with standard extensions for constraint processing. On most classes of randomly generated problems, enumeration of global revisions can even compete with local search methods concerning quick convergence on good solutions.

Enumerating global revisions is a structural approach. It performs best if even good labelings of the variables can be improved by small changes. Forthcoming applications to real world problems will have to prove whether this assumption holds on realistic constraint models. However, the framework of enumerating global revisions complies with a basic precondition for this purpose: Constraints implementing the additionally required control knowledge — local revisions of the constraints in a constraint library — can be provided as components of this library. Hence, this approach has a realistic potential to improve the state of the art in constraint processing *in practice.*

References

[1] A. Abecker, H. Meyer auf'm Hofe, J. P. Müller, and J. Würtz, editors. *Notes on the DFKI-Workshop: Constraint-Based Problem Solving*, Document D–96–02. Deutsches Forschungszentrum für Künstliche Intelligenz, 1996. `http://www.dfki.uni-kl.de/ ~aabecker/WS-CO.html`.

[2] Fahiem Bacchus and Paul van Run. Dynamic variable ordering in CSPs. In *[19]*, pages 258–275, 1995.

[3] Stefano Bistarelli, Hélène Fargier, Ugo Montanari, Francesca Rossi, Thomas Schiex, and Gérard Verfaillie. Semiring-based CSPs and valued CSPs: Basic properties and comparisons. In *[10]*, 1996.

[4] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Constraint solving over semirings. In *[13]*, pages 624–630, 1995.

[5] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5:233–270, 1992.

[6] Didier Dubois, Hélène Fargier, and Henri Prade. The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In *Proc. of the 2nd IEEE International Conference on Fuzzy Systems*, pages 1131–1136, San Francisco, CA, 1993.

[7] Didier Dubois, Hélène Fargier, and Henri Prade. Propagation and satisfaction of flexible constraints. In R. Yager and L. A. Zadeh, editors, *Fuzzy Sets, Neural Networks and Soft Computing*, pages 166–187, New York, 1994. Van Nostrand Reinhold.

[8] Eugene C. Freuder and Rick J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

[9] M. Heinrich and E.-W. Jüngst. A resource-based paradigm for the configuring of technical systems from modular components. In *CAIA–91: Proc. of the 7th IEEE Conference on AI Applications*, 1991.

[10] M. Jampel, editor. *Over-Constrained Systems.* Number 1106 in LNCS. Springer Verlag, 1996.

[11] Michael Jampel, Eugene Freuder, and Michael Maher, editors. *Workshop Notes* CP95 Workshop on Over-Constrained Systems, Cassis, France, 1995.

[12] Alan Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[13] Chris Mellish, editor. *IJCAI-95. Proceedings, 14th International Joint Conference on Artificial Intelligence*, San Francisco, 1995. Morgan Kaufmann.

[14] Harald Meyer auf'm Hofe. Representation of requirements through preference orderings of soft constraints. In *[1]*, January 1996.

[15] Harald Meyer auf'm Hofe. ConPlan/SIEDAplan: Personnel assignment as a problem of hierarchical constraint satisfaction. In *PACT-97: Proceedings of the Third International Conference on the Practical Application of Constraint Technology*, pages 257–272, London, UK, April 1997. Practical Application Company, Ltd. `http://www.dfki.uni-kl.de/ ~hmeyer/papers/pact97.ps.gz`.

[16] Harald Meyer auf'm Hofe. Finding regions for local repair in partial CSP solutions — first report. In *CP–97 Workshop on "Theory and Practice of Dynamic Constraint Satisfaction"*, Linz, October 1997. `http://www.dfki.uni-kl.de/ ~hmeyer/papers/dcsp.ps.gz`.

[17] Harald Meyer auf'm Hofe and Bidjan Tschaitschian. PCSPs with hierarchical constraint orderings in real world scheduling applications. In *[11]*, pages 69–76, 1995.

[18] Steven Minton, Mark Johnston, Andrew Philips, and Philip Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction problem and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.

[19] Ugo Montanari and Francesca Rossi, editors. *CP–95: Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, LNCS–976. Springer–Verlag, 1995.

[20] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorical algorithms: theory and practice.* Prentice Hall, Englewood Cliffs, New Jersey, 1977.

[21] Thomas Schiex, Hélène Fargier, and Gérard Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *[13]*, pages 631–637, 1995.

[22] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *AAAI-92: Proceedings of the 10th National Conference on AI*, pages 440–446, 1992.

[23] Paul Snow and Eugene C. Freuder. Improved relaxation and search methods for approximate constraint satisfaction with a maximin criterion. In *Proc. of the 8^{th} biennal conf. of the canadian society for comput. studies of intelligence*, pages 227–230, May 1990.

[24] Rick J. Wallace and Eugene C. Freuder. Conjunctive width heuristics for maximal constraint satisfaction. In *AAAI-93: Proceedings of the 11th National Conference on Artificial Intelligence, Washington DC*, pages 762–768. AAAI Press / MIT Press, 1993.

[25] Rick J. Wallace and Eugene C. Freuder. Heuristic methods for over-constrained constraint satisfaction problems. In *[11]*, pages 97–101, 1995.

List of Figures

Finding Regions for Local Repair in Hierarchical Constraint Satisfaction

Harald Meyer auf'm Hofe

RR-97-05
Research Repo